Chapitre 6
Les décorateurs

1. Introduction

Un décorateur permet d'ajouter un comportement à un élément lors de l'exécution du code. Les décorateurs prennent la forme de fonctions qui peuvent être par la suite exécutées en utilisant une expression. Celle-ci commence avec le caractère @, suivi du nom du décorateur à exécuter.

Syntaxe :

```
@DecoratorName
```

Remarque

Il n'existe pas de convention de nommage actée pour les décorateurs. Toutefois, la convention de nommage la plus communément utilisée est celle dite "PascalCase". Dans ce chapitre, c'est la convention qui sera utilisée dans les exemples.

Les décorateurs sont souvent utilisés pour appliquer des aspects techniques sur les objets, comme par exemple :

– Logger l'exécution d'une méthode.
– Injecter des dépendances.
– Notifier le changement de la valeur d'une propriété.

L'autre intérêt des décorateurs est qu'ils permettent d'ajouter des métadonnées. Celles-ci permettent de préciser des informations supplémentaires sur un élément. Elles offrent aussi la possibilité aux développeurs d'écrire du code de manière plus déclarative. Cette utilisation des décorateurs est très courante et il existe de nombreux Frameworks et bibliothèques les mettant en œuvre pour différents besoins, comme par exemple :

– Définir le verbe HTTP lié à une action d'un contrôleur (Framework/Bibliothèque de type Web MVC).
– Définir la structure d'une table en base de données (Framework/Bibliothèque de type Object-relational Mapping, ou Mapping objet-relationnel en français).
– Définir les noms de propriétés lors de la conversion d'un objet en JSON (Framework/Bibliothèque de type Parser).

Remarque

La plupart des exemples cités ci-dessus seront mis en œuvre dans le code de ce chapitre et lors du TP (cf. chapitre Un premier projet avec Node.js).

L'implémentation des décorateurs dans JavaScript est une proposition de longue date qui n'est pas encore standardisée par ECMAScript. Le concept a été introduit dans la version 1.5 de TypeScript. Cependant, les décorateurs étant considérés comme expérimentaux, le langage ne permet pas de les utiliser par défaut. Il est donc nécessaire d'activer les options de compilation dédiées lors de l'initialisation du fichier `tsconfig.json`. Lors de l'utilisation de la commande `tsc --init`, il faut ajouter les arguments `--experimentalDecorators` (qui permet d'activer l'utilisation des décorateurs dans TypeScript) et `--emitDecoratorMetadata` (qui permet d'injecter dans le code, lors de la compilation, les métadonnées autour des types. cf. section Métadonnées) pour activer le support complet des décorateurs dans TypeScript.

Exemple :

```
tsc --init --experimentalDecorators --emitDecoratorMetadata
```

Les deux options de compilation portent le même nom que les arguments dans le fichier `tsconfig.json`.

Exemple :

```
"experimentalDecorators": true,
"emitDecoratorMetadata": true
```

Remarque

Par défaut, lors de l'initialisation, ces deux options sont disponibles dans le fichier `tsconfig.json`, mais commentées.

Les décorateurs sont fortement liés à la programmation orientée objet et peuvent être appliqués aux :

– Classes
– Méthodes
– Propriétés
– Accesseurs
– Paramètres

Les décorateurs ont un ordre d'exécution qui s'applique de la dernière déclaration vers la première (on parle d'exécution *bottom to top*).

Remarque

Attention, un décorateur ne doit pas dépendre d'un ordre d'exécution. La dépendance d'un décorateur à l'exécution d'un autre est considérée comme une mauvaise pratique, car cela complexifie la maintenance d'un programme.

2. Décorateur de classe

Une fonction peut être utilisée en tant que décorateur de classe à partir du moment où son type correspond à celui de `ClassDecorator`. Ce type est défini dans le fichier de déclaration de base de TypeScript (`lib.d.ts`) :

```
declare type ClassDecorator = <TFunction extends Function>(
  target: Tfunction
) => TFunction | void;
```

Remarque

Le mot-clé `type` est utilisé en TypeScript pour définir des alias de type. Il sera abordé en détail dans la suite de cet ouvrage (cf. chapitre Système de types avancés). Le mot-clé `declare` est quant à lui utilisé dans les fichiers de définitions pour définir et typer un élément afin qu'il puisse être utilisé par la suite dans du code TypeScript.

Ce type définit que la fonction décoratrice accepte un paramètre dont le type est contraint, via l'utilisation d'un générique, à étendre celui de `Function`. Le paramètre `target` permet de récupérer le constructeur de la classe.

Exemple :

```
const LogClassName: ClassDecorator = target => {
  console.log(target.name);
};

// Log: Person
@LogClassName
class Person {
  constructor(
    public readonly firstName: string,
    public readonly lastName: string
  ) {}
}
```

Un décorateur étant une fonction, il est possible d'exécuter celle-ci directement avec la classe en paramètre.

Exemple :

```
const LogClassName: ClassDecorator = target => {
  console.log(target.name);
};

// Log: Person
LogClassName(
  class Person {
    constructor(
      public readonly firstName: string,
      public readonly lastName: string
    ) {}
  }
);
```

Remarque

Tous les types décorateurs peuvent être exécutés de cette manière. Dans la suite de ce chapitre, seule la syntaxe avec expression sera utilisée.

Lorsqu'une classe est décorée, il faut ensuite l'importer pour que le décorateur s'exécute (cf. chapitre Les modules). Si un élément est décoré, mais qu'il n'est importé dans aucun module, il est tout de même possible d'exécuter le décorateur en effectuant un import direct du module. Ce type d'import est utilisé pour déclencher un effet de bord dans le programme sans pour autant récupérer les éléments exportés par le module.

Exemple (person.ts) :

```
import "./person";
```

3. Décorateur de méthode

Une fonction peut être utilisée en tant que décorateur de méthode à partir du moment où son type correspond à celui de `MethodDecorator`. Ce type est défini dans le fichier de déclaration de base de TypeScript (`lib.d.ts`) :

```
declare type MethodDecorator = <T>(
  target: Object,
  propertyKey: string | symbol,
  descriptor: TypedPropertyDescriptor<T>
) => TypedPropertyDescriptor<T> | void;
```

Ce type définit que le décorateur accepte trois paramètres en entrée :

– `target` : l'objet dans lequel la propriété est contenue.

– `propertyKey` : la méthode qui a été décorée.

– `descriptor` : le descripteur de propriété sous la forme d'une instance du type `TypedPropertyDescriptor<T>`.

Remarque

Les descripteurs de propriété ont déjà été abordés lors de l'explication sur les boucles (cf. chapitre Types et instructions basiques).

Le type `TypedPropertyDescriptor<T>` est défini dans le fichier de déclaration de base de TypeScript (`lib.d.ts`) :

```
interface TypedPropertyDescriptor<T> {
  enumerable?: boolean;
  configurable?: boolean;
  writable?: boolean;
  value?: T;
  get?: () => T;
  set?: (value: T) => void;
}
```

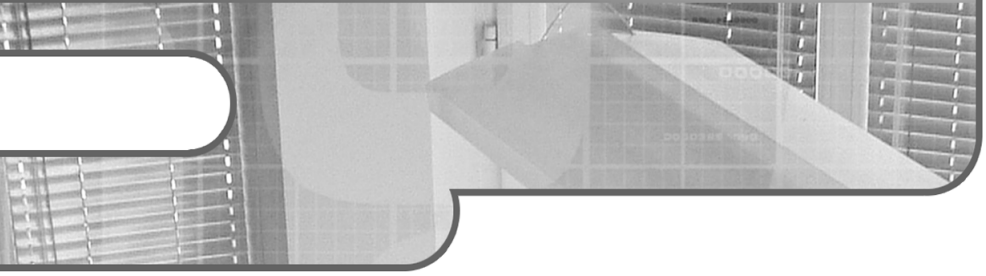
Il existe une version simplifiée de ce type qui utilise `any` à la place du type générique `T : PropertyDescriptor`. Ce type est lui aussi défini dans le fichier de déclaration de base de TypeScript (`lib.d.ts`) :

```
interface PropertyDescriptor {
    configurable?: boolean;
    enumerable?: boolean;
    value?: any;
    writable?: boolean;
    get?(): any;
    set?(v: any): void;
}
```

Remarque

Dans la section sur les boucles du chapitre Types et instructions basiques, le premier exemple montre comment définir une propriété sur un objet via l'utilisation de : `Object.defineProperty`. Le troisième paramètre attendu par la méthode `defineProperty` est typé avec l'interface `PropertyDescriptor`. Via la propriété `value` définie par cette interface, il est aussi possible de définir une méthode. C'est pour cela que les décorateurs de méthode s'appuient sur les descripteurs de propriété. De par leur nature, les décorateurs de méthode peuvent donc aussi être appliqués aux accesseurs.

Les décorateurs de méthode sont utiles pour encapsuler l'exécution d'une méthode afin de lui ajouter d'autres comportements (pouvant être appliqués avant ou après l'exécution de la méthode d'origine). Pour cela, il est nécessaire de redéfinir la méthode d'origine contenue dans la propriété `value` du paramètre `descriptor` avec une nouvelle fonction.



Chapitre 4
TypeScript

1. JavaScript

Le langage JavaScript s'est imposé depuis quelques années comme le langage indispensable au développement d'applications web.

Initialement, JavaScript a été créé comme un langage de scripting utilisé pour apporter un peu de dynamisme aux sites web statiques. Mais au fil du temps, à l'aide de la popularisation de frameworks comme jQuery ou AngularJS, l'utilisation du langage JavaScript est devenue de plus en plus importante, jusqu'à permettre la création d'applications 100 % front, possédant des fonctionnalités identiques à ce que l'on peut faire en application client lourd (mode offline, notifications, etc.).

JavaScript est un langage assez différent de ce que l'on a l'habitude de rencontrer. Tout d'abord, JavaScript est un langage interprété, le code écrit étant directement exécuté sans phase de compilation préalable.

JavaScript est également un langage dynamiquement typé, c'est-à-dire qu'un élément peut changer de type en cours d'exécution. Une variable d'un certain type peut ainsi se voir affecter une valeur d'un autre type.

```
var maFonction = function() {
    console.log("I'm a function");
}
maFonction = 27;
```

Exemple du typage dynamique JavaScript. La variable est initialisée en tant que fonction puis prend une valeur entière.

Enfin, JavaScript est un langage par nature monothreadé, fortement asynchrone et basé sur un mécanisme d'événements non bloquants. Malgré sa nature monothreadée, JavaScript est donc par design un langage très performant, permettant de tirer parti de la totalité des performances des ressources lui étant allouées (au contraire du modèle de langage multithreadé permettant de paralléliser un ensemble de traitements).

Lorsque l'on développe une application en JavaScript, sa forte dynamicité devient rapidement un handicap. Comment faire pour s'assurer que le code que l'on écrit est syntaxiquement valide ? Comment effectuer des phases de refactoring sans apporter des régressions ?

```
var maFonction = function() {
    console.log("I'm a function");
}
maFunction();
```

Dans le code précédent, il y a une erreur d'orthographe à l'appel de la méthode `maFonction`. L'erreur ne sera visible qu'à l'exécution du code.

```
var maFonction = function(param) {
    console.log("I'm a function with a param " + param);
}
maFonction();
```

Dans le code précédent, la méthode `maFonction` est appelée sans paramètre. À l'exécution, la console affichera le message `"I'm a function with a param undefined"`.

```
var maFonction = function() {
    console.log("I'm a function");
}
maFonction = 4;
maFonction();
```

Dans le code précédent, on affecte un entier à la variable `maFonction`, qui était une fonction. L'exécution de ce code renverra l'erreur suivante : `"maFonction is not a function"`.

Il existe un ensemble d'outils, comme **jslint**, permettant de valider la qualité du code écrit mais ne permettant pas de se substituer à de vraies phases de compilation.

2. TypeScript

L'une des solutions à ces problématiques a été la création de langages qui se transcompilent en JavaScript, c'est-à-dire que la compilation génère des fichiers JavaScript. C'est notamment le cas de TypeScript, porté par Microsoft et utilisé par Angular.

D'autres langages similaires existent, comme **Dart** ou **CoffeeScript**, mais TypeScript est sans doute aujourd'hui l'un des plus complets.

TypeScript est donc un langage compilé et typé fortement qui génère du JavaScript compréhensible par tous les navigateurs.

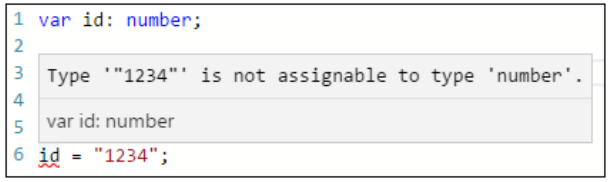
Remarque

D'autres solutions ont été trouvées, comme la création de nouveaux langages comme Flash ou Silverlight pouvant être interprétés par les navigateurs grâce à l'installation de plug-ins. Ces langages ont signé leur fin le jour où les navigateurs ont décidé de ne plus les supporter. L'avantage de TypeScript est qu'il génère du code JavaScript totalement compréhensible nativement, c'est-à-dire sans l'aide de plug-ins, par tous les navigateurs. Cette solution est donc pérenne dans le temps.

Le typage fort de ce langage va permettre d'associer un type à un élément et empêcher cet élément de changer de type. Ce typage fort permet donc une stabilité supérieure du code puisqu'il sera possible de prédire le type d'un élément, et en conséquence ses différentes valeurs possibles.

```
var id: number;
```

La phase de compilation de TypeScript permet de valider syntaxiquement le code. Si celui-ci n'est pas valide syntaxiquement, c'est-à-dire si le développeur a fait usage d'une propriété sur un objet qui n'existe pas ou qu'il utilise un type de manière non conforme, la compilation du code TypeScript indiquera cette erreur. Les phases de refactoring, qui s'avéraient extrêmement compliquées et dangereuses, sont beaucoup plus simples. De manière générale, le code sera beaucoup plus robuste et stable.



De nombreux éditeurs, notamment **Visual Studio**, **VSCode** ou **Sublime-Text** pour n'en citer que quelques-uns, comprennent le langage TypeScript et proposent de nombreux outils, comme l'autocomplétion, les erreurs de compilation directement dans l'éditeur, etc.

2.1 Syntaxe

La syntaxe de TypeScript est assez différente de ce qu'on a l'habitude de voir en JavaScript, voici un tour d'horizon des principaux éléments.

2.1.1 Variables

La déclaration d'une variable se fait de la façon suivante :

```
var maVariableBooleenne : boolean;
```

Le type est défini en suffixant la définition d'une variable par son type, séparé par ':'. Il existe un ensemble de types de base : `boolean`, `number`, `string`, `[]` pour un tableau, etc.

Dans l'optique de garder la possibilité de profiter du typage dynamique de JavaScript, TypeScript introduit le type `any`.

```
var maVariableDynamique : any;
```

Dans ce cas, TypeScript ne vérifiera pas le type de cette variable à la compilation.

Si le type d'une variable n'est pas précisé, TypeScript la considérera comme étant de type `any`.

2.1.2 Fonctions

La déclaration d'une fonction se fait de manière classique.

```
function maFonction() : void {
    ...
}
```

Le type de retour de la fonction se définit de manière identique au type d'une variable. Le type `void` indique qu'il n'y a aucun retour.

Si la fonction prend des paramètres en entrées, ces paramètres se typent de manière identique aux variables :

```
function maFonction(monParametre : string, monAutreParametre :
boolean) : void {
    ...
}
```

2.1.3 Classes

La notion de classe est apparue en JavaScript avec EcmaScript 6. Cette notion existe depuis plusieurs versions de TypeScript.

Remarque

À ce jour, la notion de classe introduite par EcmaScript 6 n'est pas encore supportée par tous les navigateurs récents, notamment par Internet Explorer 11. Il n'est donc pas encore possible de l'utiliser si l'on souhaite cibler tous les navigateurs récents.

```
class Person {
    name : string;
    age: number;

    constructor(name : string, age: number) {
        this.name = name;
        this.age = age;
```

```
    toString() {
        return `Hi I'm ${this.name} and I'm ${this.age} years old!`;
    }
}
```

Le mot-clé `class` permet d'indiquer que l'on est en train de créer une classe.

Il est également possible de créer de l'héritage entre classes :

```
class Developer extends Person {
    constructor(name, age, language) {
        super(name, age);
        this.language = language;
    }

    toString() {
        return super.toString() + ` :: I'm a Developer who likes
${this.language}`;
    }
}
```

La relation d'héritage se déclare via le mot-clé `extends`. Il est ensuite possible d'accéder aux éléments de la classe parente via la méthode `super`.

TypeScript introduit également la notion de visibilité sur les propriétés. Il est possible de déclarer une propriété `private`, qui ne sera accessible que depuis la classe courante, `protected`, qui ne sera accessible que depuis la classe courante ou les classes héritant de la classe courante, et enfin une propriété `public` qui sera accessible depuis l'extérieur de la classe. Par défaut, si aucune visibilité n'est spécifiée, une propriété sera `public`.

```
class Modifier {
    public myPublicProperty: string;
    protected myProtectedProperty: string;
    private myPrivateProperty: string;
}
```